Mystery Bug Theater

Renée Bryce and Vicki Allan
Utah State University
Logan, UT 84321
Renee.Bryce, Vicki.Allan@usu.edu

Abstract

Introductory Computer Science students often encounter programming bugs. Our previous work gathers and classifies data for 450 programming bugs brought to our tutor lab over a one year period. We use the data to identify the most common bugs as the basis for activities that improve our curriculum. This paper discusses several activities that rely on this data: (1) the "Mystery Bug Theater" website contains games and movies about common bugs, (2) professors quickly respond to the common bugs in lectures, and (3) class exercises use buggy code from the repository. Future work will distribute software testing material across the curriculum to help students with the most common bugs in specific courses and will analyze whether bug patterns change based on curriculum changes.

1. Introduction

Despite the growing need for Computer Science graduates, the United States has experienced shrinking enrollments of students in this major [7]. The Utah State University Computer Science department is representative of these national trends. While there are many factors that contribute to enrollment and retention, our work focuses on just one factor, that of understanding and addressing student programming bugs. We examine this specific issue because a recent survey that we administered reveals that 42% out of 64 of our freshman respondents identify "debugging" or "programming bugs" as the top frustration that they have in our introductory programming course. This is particularly informative because we intentionally asked about their biggest frustration, but asked it in an open ended manner (i.e., a text field) so that we would not bias the responses by providing specific categories. The category of "debugging/programming bugs" emerged as the top problem. This response motivates us to better understand the programming bugs that students find difficult and to develop solutions.

We need to develop solutions that help future students to avoid common bugs. Our current goals are to: (1) share the data with professors so that they can rapidly address problems that cause students to bring bugs to the tutor lab, (2) develop in-class exercises that are tailored to the common bugs that we observe in specific courses across the curriculum, and (3) create a website with movies and games that help students to learn about common bugs that they can avoid. Section 2 briefly reviews our data collection and classification process, Section 3 provides examples of our current games, movies, and curriculum work. Section 4 discusses related work, and Section 5 discusses conclusions and future work.

978-1-4577-0348-5/11/$26.00 CSEE&T 2011, Waikiki, Honolulu, HI, USA

2. Background

Our previous research describes our data collection and analysis [1]. We briefly review this previous work as it feeds into the design of our website, http://digital.cs.usu.edu/∼bugs.

2.1. Data collection process

At Utah State University, our tutor lab is staffed by upper division students and is open 6 days a week. Students pay a required fee for Computer Science courses which covers the cost of the tutor lab. Students complete an online form before receiving help from the tutor that includes their class year, course (i.e., CS1, CS2), the programming language, number of lines of code, and the amount of time that they spent trying to solve the problem before asking a tutor for help. Next, they provide a brief description of the assignment and the problem that they encountered. The tutor then sits with the student to review the information that they entered into the web form. Once this is done, the tutor walks through the code with the student and helps them to understand their problems. During the tutor session, students sometimes find that they described the problem incorrectly or that there were additional problems that they did not know before the tutor helped them. This is clarified in the second step of our data collection process. After the tutor helps the student, they work together to fill out a web form to summarize the bug(s), the solution(s), and optionally provide one or more test cases that would have exposed their bug(s). We store the data on our web server.

2.2. Analysis

Our previous and current work analyzes 450 bugs over the course of one year and classifies them into 20 categories. Due to space limitations, we refer the reader to [1] for a full listing and description of the 20 categories and the frequency of bugs that fall into these categories. We instead briefly summarize the categories for the top 5 most common bugs. **Problem solving** was the most common bug for 113 students. One example is a student working on an assignment to implement a binary search tree that does not understand the tree terminology such as, root, left child, right child, or leaf. They also need the tutor to trace through an example of a tree traversal algorithm before they are prepared to solve their binary search tree problem. **File input/output (i/o)** is the second most common problem experienced by 63 students. The majority of these students struggle with syntax problems. **Loop** bugs are encountered by 41 students. Many of these bugs are the result of off-by-one mistakes or infinite loops that do not have a stopping case. **Pointers** cause bugs for 33 students, many of which have difficulty with null pointers. **Array** bugs occurred for 31 students. Off-by-one mistakes are a common problem in this category. Again, we refer the reader to [1] for more details on this data.

3. Bug-based Activities

The data that we collect provides opportunities to improve our curriculum. In this section, we discuss our broad collection of solutions that we are actively exploring.

3.1. Pinpointing Bugs Quickly

We regularly collect and classify data that allows us to send "bug updates" to instructors. Instructors are then able to quickly address common problems. Lectures can be augmented with material to clarify problem areas revealed by our bug analysis reports. Our future work will survey

Figure 1. Example game: Binary Trees

instructors about how they respond to this data, including how often they use the data (i.e., should we send bug analysis reports according to a schedule or when a specific count of students encounter similar bugs) and how they addressed the bugs in lecture.

3.2. Class Exercises Based on Buggy Code

We provide a set of exercises where students identify bugs in code snippets that we collected from previous students. We then discuss each bug and how to avoid similar bugs in the future. One example code snippet that we use does not check whether a pointer is null. A second example is convoluted by nondescriptive variable names. This exercise is currently in use in our CS3 course. We are extending our collection of buggy code for this exercise by pulling real examples from our continually growing repository of bugs.

3.3. Software Testing Across the Curriculum

A collection of software testing lectures and exercises will be spread across our curriculum based on the bugs reported in the different courses. For instance, we emphasize that test cases that achieve def-use coverage are useful in CS3 because we observe that students often have extra variables that cause confusion and bugs. This is particularly problematic in their assignments that use recursion. We also notice that test cases that achieve branch coverage are useful for these students, who often create unreachable code due to return statements in both branches of a conditional.

3.4. Mystery Bug Theater

We regularly update the "Mystery Bug Theater" website with statistics about the bugs. We also use this data to select common bugs to use as topics in our games and movies.

Games: We experiment with several formats for games. For instance, a first game asks multiple choice questions. If the student answers the question incorrectly, they are asked to try again. If their answer is correct, we summarize why their answer is correct, present data about the number of students that experienced that bug in a previous semester, and add points to their score. A second game shown in Figure 1 asks the student to label a diagram such as the nodes of a tree. If the answer is incorrect, they are encouraged to fix their mistakes. If the answer is correct, they are able to move on to the next question. We currently have the option for students to rate these games on our website and our future work will perform usability testing with a focus group of students so that we can identify useful examples and formats for future games.

Movies: The current movie format used on our website plays slides that discuss the common bugs and how to avoid them. Similar to our games, we will use a focus group of students to perform usability testing.

3.5. Analysis of bugs over time

Finally, the timeliness of our data allows us to relate bugs to course characteristics such as book, language, and IDE. Our faculty continuously improve our courses and assess the changes. For instance, we may see bugs that are more specific to a programming language or IDE. The bug data that we collect provides an extra feedback loop that complements the analysis of instructor opinions, student assessment scores, and teacher evaluations. We anticipate that instructors will be able to query our database to view bugs by finer details such as course, language, IDE, or category of bug. We have not yet addressed this final issue of relating bugs to course characteristics, but we envision this as part of our future work.

4. Related work

Several approaches exist to collect student programming bugs. We briefly review a few examples of such work. Fenwick et. al. [3] use their ClockIt Data Logger to track student coding patterns. ClockIt records data such as time between compilations, object instantiation and invocations, and compiler errors. Their ClockIt experiments find that unknown variables, unknown methods, and missing semicolons rank as the top three errors. These results were different than similar work by Jadud [5] in which they report missing semicolons, unknown variables, and missing brackets as the top three errors for their students respectively. In addition to automated approaches for data collection, Ko et. al., collect data through observation of users that program in Alice. They study the cognitive causes of programming bugs [6]. Our work differs from these previous works as we do not record coding patterns or observe students, but rather collect data through a web form where students and tutors sit down together to enter descriptions of the bugs that they bring to our tutor lab. The students describe their bugs "before and after" their interactions with a tutor in our tutor lab. The student records (1) their personal understanding of their bug(s) before a tutor helps them and then (2) their personal understanding of the problem and solution after the tutor helps them. This data provides a different view point from previous studies as our students that visit the tutor lab document problems that they feel that they can not solve on their own. Of course, while this data provides a different viewpoint, we acknowledge threats to validity because we only gather data from students that seek help from our tutor lab, we rely on the honesty and accuracy of students to report data, and our data may not be representative of the types and frequency of errors encountered by students at other universities.

In addition, several websites exist that provide animations of Computer Science topics. To our knowledge, the only interactive website that covers software bugs is BugHunt [2]. BugHunt differs from our work as it does not collect or report data about student programming bugs, but rather interactively teaches students about software testing topics and provides exercises for students to find bugs in code. They provide interactive exercises on topics such as black box testing and code coverage.

Finally, several bug classification schemes exist. (See [6] for a summary.) Our current classification scheme is based on our specific data. It has evolved over time as we have seen new types of bugs from different course offerings. To avoid an *ad hoc* classification scheme in the future, we are moving to the IEEE 1044-1993 Standard Classification for Software Anomalies [4].

5. Conclusions and Future Work

In our previous and current work, we identify the most common programming bugs that students bring to our tutor lab. We now use this data for several purposes.

First, we provide the data to professors so that they can quickly address problems that many students struggle with on assignments. Instructors are able to modify lectures when they learn that a large number of students are struggling with a specific type of bug. Our future work will survey instructors on the changes that they make throughout the semester in response to the data.

Second, we create exercises for students to analyze buggy code that previous students have written. Our bug collection process allows us to continually add new exercises.

A third goal is to incorporate software testing material throughout our curriculum. We will tailor the distribution of material based on the bugs that are most common in different courses. For instance, certain types of bugs are likely to be revealed if students write test cases that focus on branch coverage; if students work on equivalence classes for test cases; or if students use mutation testing. We will collect more data to help us choose the initial distribution of software testing material. Gathering data from different semesters offsets bias that may occur when different instructors teach the courses.

Next, we created an initial prototype of the "Mystery Bug Theater" website which has a theme to share data about the common bugs and to provide games and movies that help future students to avoid those bugs. We will encourage students to view the data and play the movies and games so that they can preemptively avoid the most common bugs from previous students in their same courses. Prior to this release, our ongoing and future work will perform usability testing on our current movie and game content and formats.

Finally, our faculty often vote on changes to *improve* different courses. We will monitor how the bugs change when we modify course characteristics such as textbook, language, or IDE. Indeed, assessing how bugs change involves many complicated issues and is an area of future work.

6. Acknowledgments

Thank you to the undergrad research alumnae from 2009-2010, Alison Cooley, Amy Hansen, and Nare Hayrapetyan, who contributed to the Mystery Bug Theater website.

References

[1] R. Bryce, A. Cooley, A. Hansen, and N. Hayrapetyan. A one year empirical study of student programming bugs. In *Proceedings of the Frontiers in Education Conference (FIE)*, pages 122–127. IEEE, October 2010.

[2] S. Elbaum, S. Person, J. Dokulil, and M. Jorde. Bug hunt: Making early software testing lessons engaging and affordable. In *29th International Conference on Software Engineering (ICSE'07)*, pages 688–697. ACM, May 2007.

[3] J. Fenwick, C. Norris, F. Barry, J. Rountree, C. Spicer, and S. Cheek. Another look at the behaviors of novice programmers. In *Proc. of the SIGCSE conference on Computer Science Education*, pages 296–300. ACM, March 2009.

[4] I. S. D. W. Group. *1044-2009 - IEEE Standard Classification for Software Anomalies*. IEEE, New York, January 2010.

[5] M. C. Jadud. Methods and tools for exporing novice compilation behaviour. In *Proc. of the International Computing Education Research Workshop (ICER)*, pages 73–84. ACM, 2006.

[6] A. J. Ko and B. A. Myers. Development and evaluation of a model of programming errors. In *Symposia on Human-Centric Computing Languages and Environments*, pages 7–14. IEEE, 2003.

[7] J. Vegso. Freshman interest in cs and degree production trends. Computing Research Association (CRA), Accessed on January 21, 2011. `http://www.cra.org/wp/index.php?p=126/`.